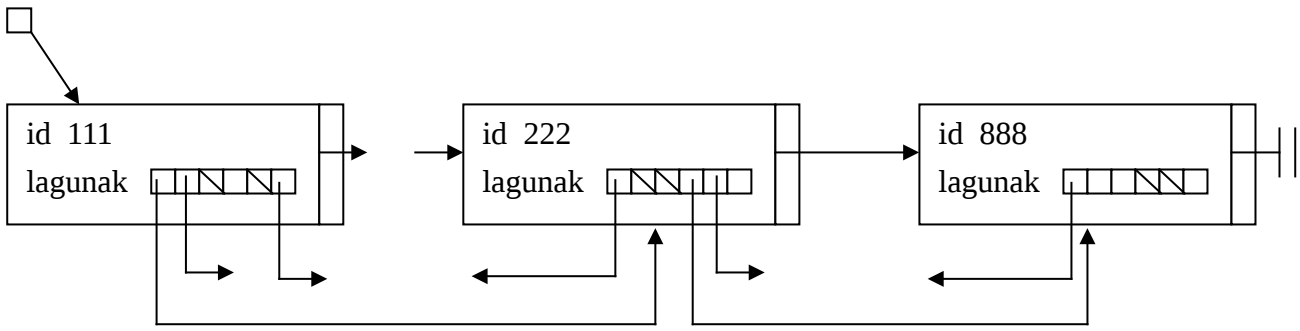
***ARIKETA 1.- Lagunen zerrenda (1,5 puntu)***

Datu-egitura bat definitu da Lista DMA adierazteko. Lista horretan pertsonak gordeko dira, elementu batetik bere lagun guztiei atzipen zuzena egongo delarik. Zerrenda gorantz ordenatuta dago, pertsonen identifikatzailearen arabera.

Pertsona batek gehienez 10 lagun izan ditzake. Adiskidetasuna simetrikoa da, hau da, A Bren laguna baldin bada, orduan B Aren laguna izango da.

```
public class Pertsona {
   String id;
   Pertsona[] lagunak; // 10 elementu: bere lagunak (null izan daitezke)
}

public class Adabegi {
   Pertsona info;
   Adabegi next;
}

public class Lista {

   Adabegi first;

   public void gehitu(String id, String[] kideak){
   // Aurrebaldintza: “id” ez da listako elementu bat
   //                 “kideak” zerrendako elementuak listan daude
   // Postbaldintza: “id” balioari dagokion elementua zerrendan gehitu da.
   //                Gehitu dira “id” elementuari
   //                zegozkion loturak (bi norabideetan)
}
```

Adibidez:

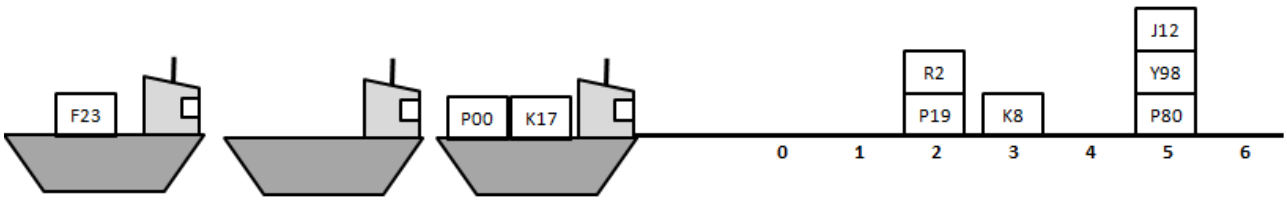



*gehitu(id, kideak)* algoritmoa **inplementatu** eta bere **kostua kalkulatu**.

Aurreko adibidean, gehitu*(“333”, <111”, “222”, “888”>)* deiak elementu hori zerrendan gehituko luke (“222” elementuaren ondoren) eta, gainera, “111”, “222” eta “888” elementuekin loturak ere gehituko lituzke.

**2. ariketa: Portua (1,5 puntu)**

Portu bateko jardueraren simulazioa egin nahi dugu. Portuak hainbat nasa ditu kontainerrak pilatzeko. Eguna hasten denean, ontziak ilaran sartzen dira portura, nasetatik kontainerrak kargatzeko edo nasetan kontainerrak deskargatzeko.




(ontziak) (nasak)

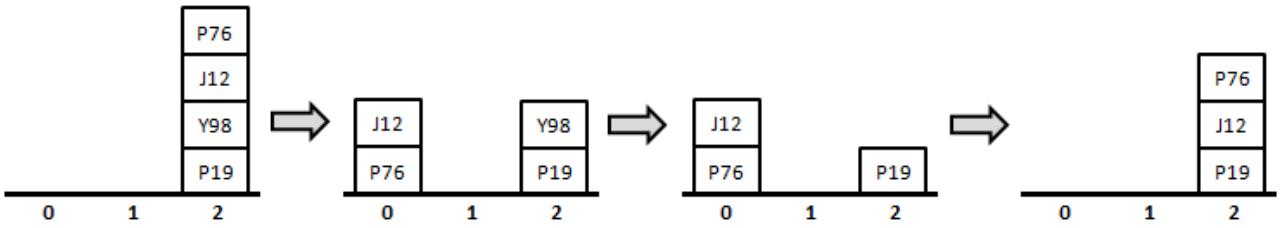
Ontzi bakoitza mota batekoa da (kargako ontzia edo deskargako ontzia) eta hainbat eskaera ditu betetzeko. Eskaera bakoitzak kontainer baten kodea eta nasa baten zenbakia ditu. Deskargako ontziek nasetan kontainerrak deskargatzeko eskaerak izango dituzte, eta kargako ontziek nasetatik kontainerrak kargatzekoak. Badakigu ere 0 zenbakia duen nasa berezia dela eta ez dela eskaeretan agertuko.

Horrela funtzionatuko du jardueraren simulazioak, ontzien ilaran ontziak dauden bitartean:

- Ilarako lehen ontzia hartuko da.

- Deskargako ontzi bat bada:
  - Ontziaren lehenengo *maxEskaera* eskaerak kudeatuko dira (gutxiago baditu, dituen guztiak):
    - Eskaera bakoitzeko: kontainerra adierazitako nasan utziko da.
    - *maxEskaera* eskaera baino gehiago bazituen, ontzia ontzien ilaran jarriko da berriz, ilararen amaieran.

- Kargako ontzi bat bada:
  - Ontziaren eskaera guztiak kudeatuko dira:
    - Eskaera bakoitzeko: kontainerra adierazitako nasatik kendu behar da. Kontainer bat kendu behar denean, 0. nasa erabili dezakegu aldi baterako bere gainean dauden kontainerrak uzteko, eta behin gure kontainerra kendu dugula, 0. nasan utzi ditugunak jatorrizko nasan utziko ditugu berriz. Adibidez, ondorengo irudiko Y98 kontainerra kentzeko:

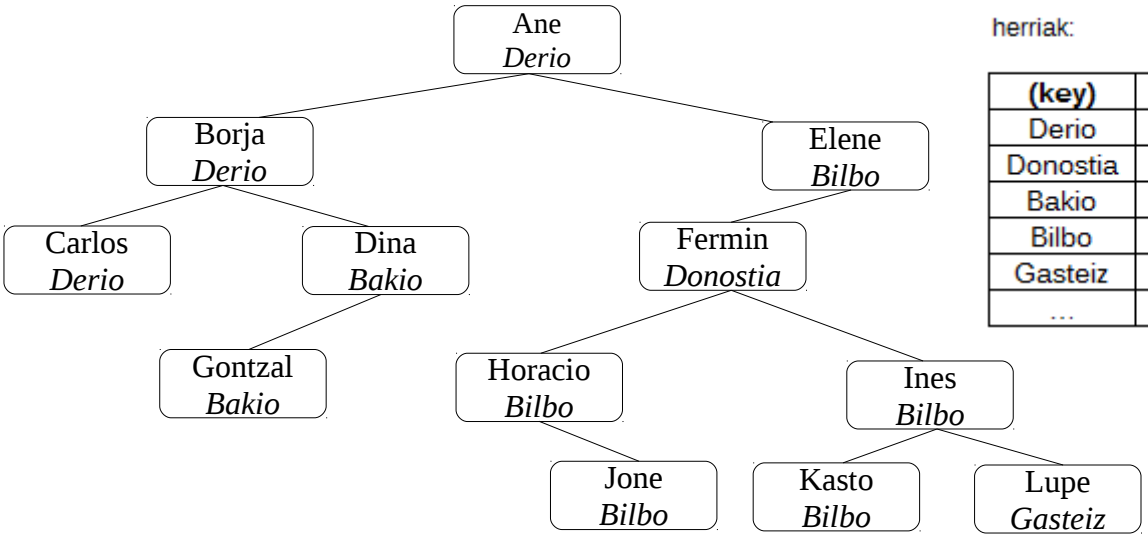



*Portua* klasearen *portuaSimulatu* metodoa inplementatzea eskatzen da.

```
public class Eskaera {
	String kontainerKode;
	int nasa;
}

public class Ontzia {
	int mota; //0 (deskarga), 1 (karga)
	String izena;
	Queue<Eskaera> eskaerak;
}

public class Portua {

	Stack<String> nasak[];

	public void portuaSimulatu(Queue<Ontzia> ontziak, int maxEskaera, int nasaKop){
	//Aurre: maxEskaera deskargako ontzi batek txanda batean kudeatu ditzakeen
	//eskaera kopurua da (>=1)
	//Aurre: nasaKop portuaren nasa kopurua da (>=2). 0. nasa berezia da.
	//Post: portuaren jarduera simulatu da, ontzien eskaerak betez.


		//Nasak sortu

			...


		//Portuaren jarduera simulatu

			...



	}
}
```

**3. ariketa: Zuhaitz genealogikoa (1,5 puntu)**

Zuhaitz genealogikoa adierazteko zuhaitz bitar bat erabili dezakegu. Adabegi bakoitzak pertsona bat gordeko du, bere izena eta jaioterriarekin. Gainera adabegi bakoitzaren ezkerreko umeak pertsona horren aita nor den adieraziko du, eta eskuineko adabegiak pertsona horren ama. Adibidez, irudiko zuhaitzean, Ane Derion jaio zela eta bere gurasoak Borja eta Elene direla ikus dezakegu. Zenbait kasutan informazioa ez da osorik egongo (adibidez ez dakigu nor den Eleneren ama).




herriak:

| (key) | (value) |
|---|---|
| Derio | Bizkaia |
| Donostia | Gipuzkoa |
| Bakio | Bizkaia |
| Bilbo | Bizkaia |
| Gasteiz | Araba |
| ... | ... |

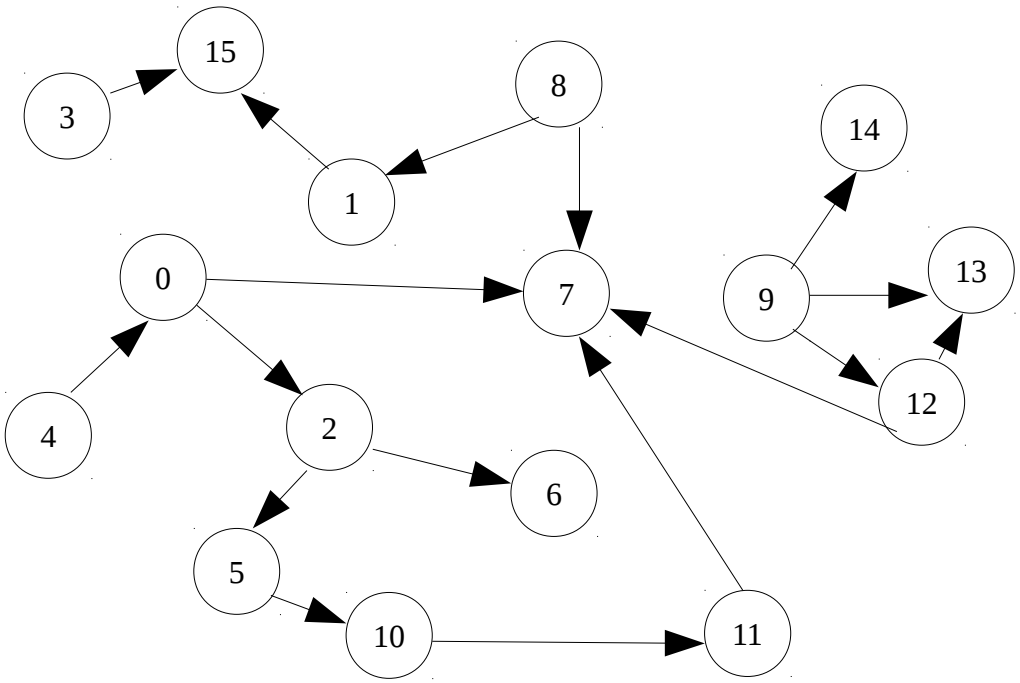
*bizkaitarPetoPetoaDa* metodoa inplementatu behar duzue. Metodo honek bi parametro jasoko ditu: pertsona baten izena eta HashMap bat. Irudian ikus daitekeen moduan, HashMap-ean hainbat herriren izenak edukiko ditugu, bakoitza zein probintziatan dagoen adieraziz. Metodoak **pertsona hori bizkaitar peto-petoa den** kalkulatu behar du (eta **kostua adierazi** behar duzue). Pertsona bat bizkaitar peto-petoa izango da baldin eta honakoak betetzen badira:
- Zuhaitzean dago (suposatuko dugu zuhaitzean ez dagoela pertsona-izen errepikaturik)
- Bizkaiko herri batean jaio zen.
- Ezagunak diren bere arbaso guztiak ere Bizkaiko herri batean jaio ziren.

Adibidez, Borja bizkaitar peto-petoa da, bera eta bere arbaso ezagun guztiak (Carlos, Dina eta Gontzal) Bizkaiko herritan jaio zirelako. Ane berriz ez da bizkaitar peto-petoa, bere arbaso batzuk (Fermin eta Lupe) ez zirelako Bizkaian jaio.

```
public class Pertsona {
  String izena;
  String jaioterria;
}
public class BinaryTreeNode<T> {
  T data;
  BinaryTreeNode<T> left, right;
}
public class ZuhaitzGenealogikoa {
  BinaryTreeNode<Pertsona> root;

  public boolean bizkaitarPetoPetoaDa(String izena,HashMap<String,String> herriak){
                ...
  }}
```

**4. ariketa: Kontakturik lagungarriena (1,5 puntu)**

Ondorengo grafoan adabegiek pertsonak adierazten dituzte, eta arkuek pertsona bakoitzak sare sozialetan dituen kontaktuak.




Pertsonak batek transplante bat behar duenean sare sozialetan mezu bat jarriko du berarekin bateragarriak diren pertsonen bila (bere kontaktuek ikus dezaten) eta kontaktuei eskatuko die mezua era berean hedatzeko.

***lagungarriena*** **metodoa inplementatu** eta bere **kostua adierazi** behar duzue. Metodo honek transplante bat behar duten pertsonez osaturiko ArrayList bat jasotzen du, eta zerrenda horretan dauden pertsonekiko aldi gehienetan lagungarria izan daitekeen grafoko pertsona bilatu behar du (berdinketa badago, indizerik baxuena duena). X pertsona bat lagungarria izan daiteke Y pertsonarekiko X eta Y bateragarriak badira eta Y-k sare sozialetan X-en mezua jasotzen badu, kontuan izanda mezuak goian adierazitako eran hedatzen direla. Bi pertsona bateragarriak diren edo ez jakiteko *bateragarriakDira* metodoa erabili behar duzue (ez da inplementatu behar).

```
public class KontaktuSarea {
        private boolean[][] adjMatrix; //adjacency matrix

     public int lagungarriena(ArrayList<Integer> pertsonak){
    . . .
    }
    private boolean bateragarriakDira(Integer p1, Integer p2){
    //POST: p1 eta p2 bateragarriak diren itzultzen du. EZ DA INPLEMENTATU BEHAR
    }
}
```

Adibidez, sarrerako zerrenda <4, 8, 2, 9> baldin bada, eta hauek baldin badira bikote bateragarriak: (4, 7), (4, 10), (2, 10), (8, 7), (8, 10) eta (9, 7).
Adibide horretan 7 izango litzateke lagungarriena (4tik, 8tik, eta 9tik atzigarria eta bateragarria, hau da, 3 aldiz lagundu dezake). 10 zenbakiko adabegia, adibidez, bakarrik bi elementurentzat da lagungarria (4rentzat eta 2rentzat).

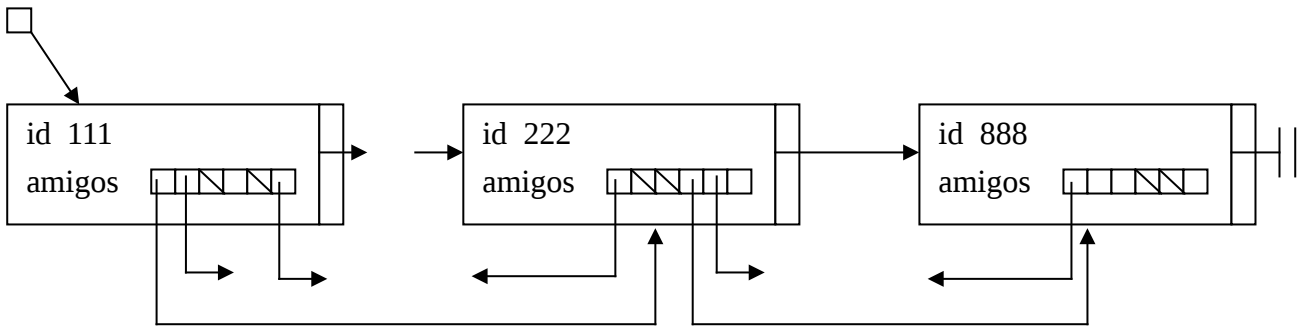
***EJERCICIO 1.- Lista de amigos (1,5 puntos)***

Se ha definido una estructura de datos para implementar el TAD Lista, donde una lista contiene una serie de personas, permitiéndose acceder desde una persona a todos sus amigos de manera directa. La lista está ordenada ascendentemente por el identificador de la persona.

Una persona puede tener como máximo 10 amigos diferentes. La relación de amistad es simétrica, es decir, si A tiene a B como amigo, entonces B tiene como amigo a A.

```
public class Persona {
   String id;
   Persona[] amigos; // 10 elementos: sus amigos (null si no apunta a nadie)
}

public class Nodo {
   Persona info;
   Nodo next;
}

public class Lista {

   Nodo primero;

   public void añadir(String id, String[] colegas){
   // Precondicion: "id" no está en la lista
   //               todos los elementos de "colegas" están en la lista
   // Postcondicion: se ha añadido el elemento correspondiente a "id".
   //               También se han añadido las referencias
   //               de los amigos de la lista "colegas"
}
```

Ejemplo:

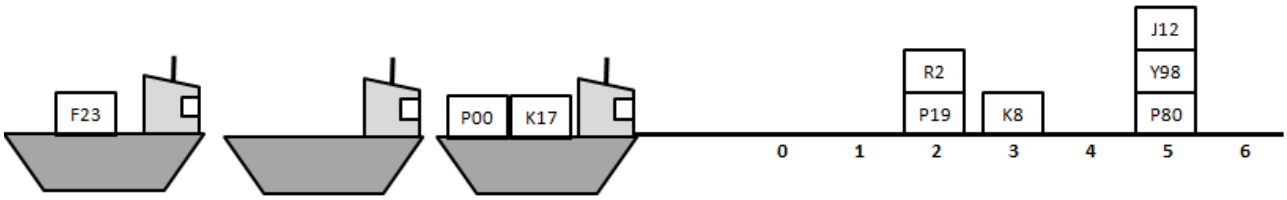



Se pide **implementar** el subprograma añadir*(id, listaDeColegas)* y **calcular el coste** del algoritmo.

En el ejemplo anterior, la llamada a añadir*("333", <"111", "222", "888">)* añadiría el elemento "333" a la lista, después de "222". Además, también añadiría los enlaces a y desde "333" con "111", "222" y "888".

**Ejercicio 2: Puerto (1,5 puntos)**

Queremos simular la actividad de un puerto. El puerto tiene diferentes muelles para guardar los contenedores que llegan. Cuando empieza el día, los barcos entran ordenadamente al puerto para cargar y descargar contenedores a y desde los muelles.




(barcos) (muelles)

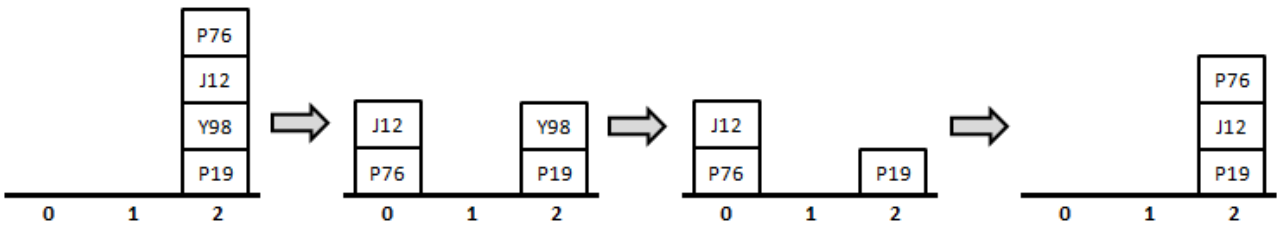
Cada barco es o bien de carga o de descarga, y tiene una serie de peticiones a completar. Cada petición tiene un código de contenedor y un número de muelle. Los barcos de descarga tendrán peticiones para descargar contenedores en el muelle, y los de carga para cargar contenedores desde el muelle. El muelle de código 0 es especial y no aparecerá en las peticiones.

La simulación de la actividad del puerto funciona de la siguiente manera, hasta que se atiendan todas las peticiones de los barcos:

- Se atenderá el primer barco que ha llegado.

- Si es un barco de descarga:
  - Se atenderán las primeras *maxPeticiones* de un barco (si tiene menos, todas)
    - Para cada petición: se dejará el contenedor en el muelle especificado.
    - Si el barco tuviera más de *maxPeticiones*, se colocará el barco al final de la cola con las peticiones restantes.

- Si es un barco de carga:
  - Se atenderán todas las peticiones del barco:
    - Para cada petición: el contenedor se cogerá del muelle especificado. Para sacar un contenedor, se puede usar el muelle 0 para dejar ahí temporalmente los contenedores que se encuentran encima del contenedor pedido y, una vez quitado ese contenedor, se volverán a colocar en ese muelle los contenedores apilados temporalmente en el muelle 0. Por ejemplo, en la siguiente figura se muestra cómo se cargaría en el barco el contenedor Y98:

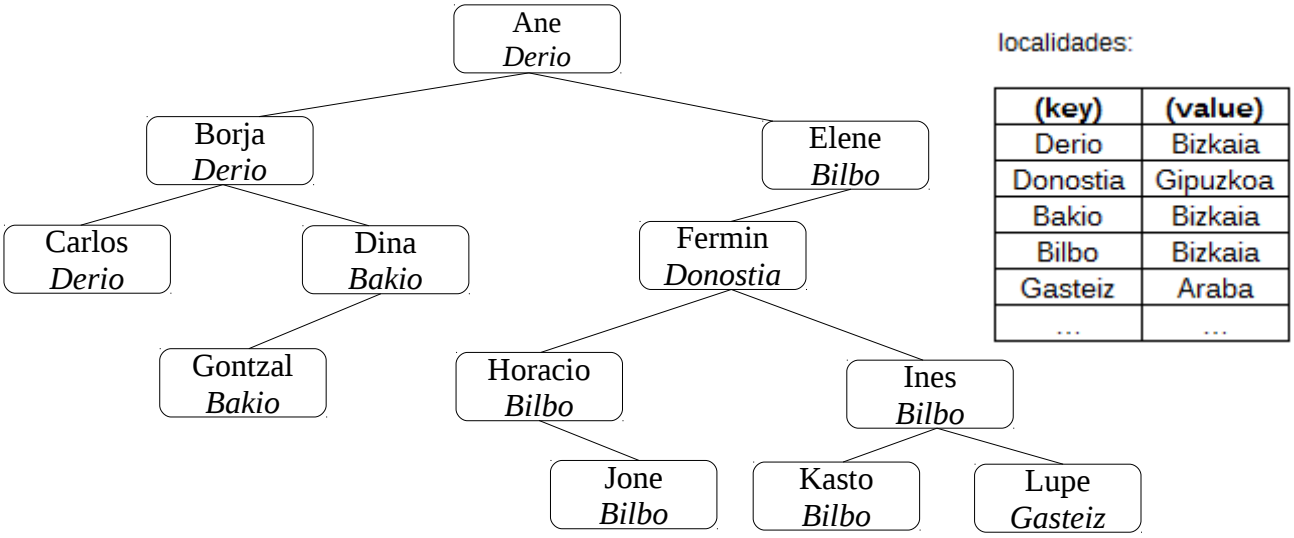



Se pide implementar el método *simularPuerto* de la clase *Puerto.*

```
public class Peticion {
	String codigoDeContenedor;
	int muelle;
}

public class Barco {
	int tipo; // 0 (descarga), 1 (carga)
	String nombre;
	Queue<Peticion> peticiones;
}

public class Puerto {

	Stack<String>[] muelles;

	public void simularPuerto(Queue<Barco> barcos, int maxPeticiones, int numMuelles){
	// Pre: maxPeticiones es el número máximo de peticiones que se pueden atender en
	//	el turno de un barco
	// Pre: numMuelles es el número de muelles del puerto (>=2). El muelle 0 es especial
	// Post: se ha simulado la actividad del puerto, atendiendo las peticiones de los barcos


		// crear los muelles

			...


		// simular la actividad del puerto

			...




	}
}
```

**Ejercicio 3: Árbol genealógico (1,5 puntos)**

Para representar un árbol genealógico podemos usar un árbol binario. En cada nodo se guardará una persona, que tendrá nombre y localidad de nacimiento. Además, el hijo izquierdo de un nodo indicará quién es el padre de dicha persona, y el hijo derecho quién es la madre. Por ejemplo, en la imagen podemos ver que Ane nació en Derio, y que sus padres son Borja y Elene. En algunos casos no dispondremos de información completa (por ejemplo no sabemos quién es la madre de Elene).




localidades:

| (key) | (value) |
|---|---|
| Derio | Bizkaia |
| Donostia | Gipuzkoa |
| Bakio | Bizkaia |
| Bilbo | Bizkaia |
| Gasteiz | Araba |
| … | … |

Debéis **implementar el método *esVizcainoDePuraCepa*** e **indicar su coste**. El método recibe dos parámetros: el nombre de una persona y un HashMap. Como puede verse en la imagen, el HashMap contiene el nombre de varias localidades junto con su provincia. El método debe calcular si dicha persona es vizcaína de pura cepa. Una persona será vizcaína de pura cepa si se cumplen las siguientes condiciones:

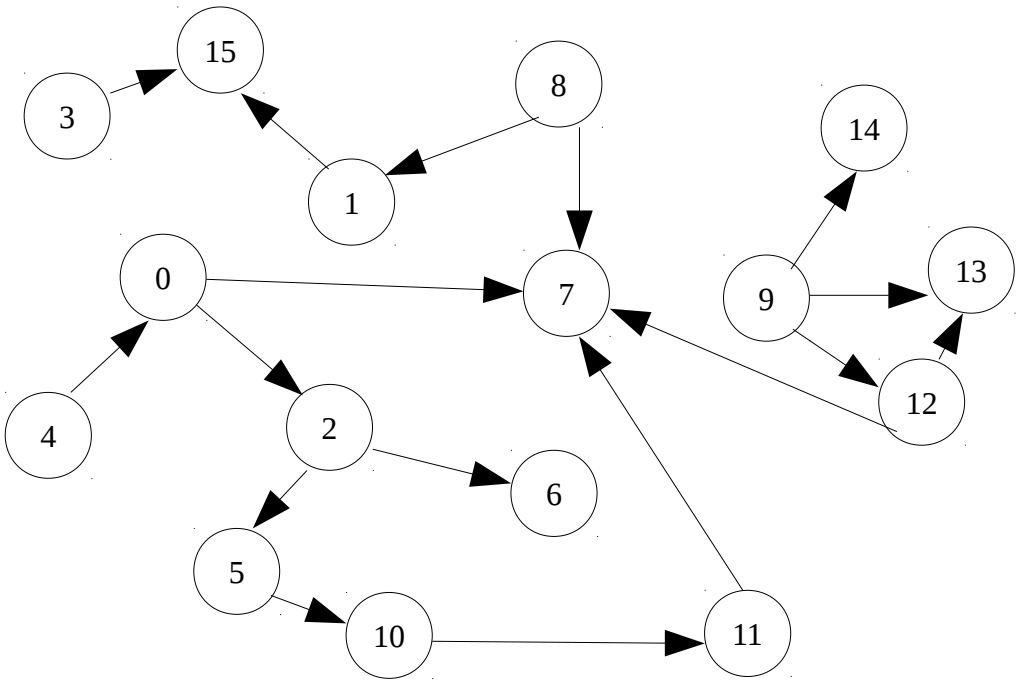
- Está en el árbol (supondremos que en el árbol no hay nombres de persona repetidos)
- Nació en alguna localidad de Bizkaia.
- Todos sus antepasados conocidos nacieron en alguna localidad de Bizkaia.

Por ejemplo Borja es vizcaíno de pura cepa porque tanto él como todos sus antepasados conocidos (Carlos, Dina y Gontzal) nacieron en localidades de Bizkaia. Ane sin embargo no es vizcaina de pura cepa porque algunos de sus antepasados (Fermin y Lupe) no nacieron en Bizkaia.

```
public class Persona {
  String nombre;
  String localidadNacimiento;
}
public class BinaryTreeNode<T> {
  T data;
  BinaryTreeNode<T> left, right;
}
public class ArbolGenealogico {
 BinaryTreeNode<Persona> root;

 public boolean esVizcainoDePuraCepa(String nombre,HashMap<String,String> localidades)
 {
   ...
 }
}
```

**Ejercicio 4: El contacto que más puede ayudar (1,5 puntos)**

En el siguiente grafo, los nodos representan personas y los arcos indican los contactos de cada persona en redes sociales.




Cuando una persona necesite un transplante, pondrá un mensaje en sus redes sociales para buscar personas compatibles con ella, de manera que sus contactos puedan ver este mensaje y difundirlo a otros contactos, y así sucesivamente.

**Debéis implementar el método *elQueMasPuedeAyudar* e indicar su coste**. Este método recibe un ArrayList con personas que necesitan un transplante, y debe buscar la persona del grafo que puede ayudar a más personas de dicha lista (en caso de empate la de menor índice). Una persona X podrá ayudar a una persona Y si X e Y son compatibles para el transplante y además X recibe por redes sociales el mensaje de Y, sabiendo que los mensajes se difunden según lo explicado en el párrafo anterior. Para saber si dos personas son compatibles para un transplante debéis utilizar la función *sonCompatibles* (no hay que implementarla).

```
public class RedDeContactos {
    private boolean[][] adjMatrix; //adjacency matrix

     public int elQueMasPuedeAyudar(ArrayList<Integer> personas){
    . . .
    }
    private boolean sonCompatibles(Integer p1, Integer p2){
    //POST: Devuelve si p1 y p2 son compatibles. NO HAY QUE IMPLEMENTARLA
    }
}
```

Por ejemplo, supongamos que la lista de entrada es <4, 8, 2, 9> y que éstas son las parejas compatibles: (4, 7), (4, 10), (2, 10), (8, 7), (8, 10) y (9, 7).

En este ejemplo 7 sería el que más puede ayudar (porque es accesible desde 4, 8 y 9 y compatible con ellos, es decir, puede ayudar a 3 contactos). La persona con el número 10 sin embargo sólo puede ayudar a dos personas (4 y 2).